Toward a Grid Technology Independent Programming Interface for HEP Applications

S. Campana, A.Sciabà, Antonio Delgado Peris, F. Donno, P. Méndez Lorenzo, R. Santinelli
CERN, Geneva, Switzerland

Abstract

In the High Energy Physics community, Grid technologies have been accepted as solutions to the distributed computing problem and several Grid projects have been deployed. Unfortunately, the programming interfaces presented to the end user are often not uniform or provide different levels of abstractions. Furthermore, Grid technologies are constantly evolving and the changes in the programming interfaces make it hard for the end user to build Grid applications. After analyzing the existing programming interfaces for the LHC Computing Grid middleware, we identified some ways to provide high-level technology independent interfaces. In this article, we will propose a prototype for high-level interface to a security service, focusing on the authentication process.

INTRODUCTION

As high energy physics evolves, both theoretical and experimental studies become extremely computing intense and demanding. One good example is the physics at the Large Hadron Collider (LHC), located at CERN and starting data acquisition in 2007. It is expected that the LHC experiments will generate about 13 PB of data every year, requiring the equivalent of 70000 of today's fastest PC processors.

To satisfy such demand for computing resources, the community agreed on a geographically distributed Computational Data Grid, which can integrate the capacity of several computing centers into a virtual computing organization. For this purpose, the LHC Computing Grid (LCG), especially aimed at HEP applications, provides a set of services and interfaces, both in the form of command line tools as well as Application Programming Interfaces (API) in C, C++, Java, etc.

While the LCG middleware has been largely deployed and is in production since quite some time, still few Grid-enabled applications exist that exploit the full potential of the Grid. In fact, one of the most serious difficulties that a user encounters when developing a Grid application is the lack of a consistent set of programming interfaces to all Grid services. In addition, different implementations of similar Grid services present different interfaces and do not easily interoperate. One of the reasons is the rapid evolution of the middleware, as new functionalities are often added and the existing ones continuously improved. Because of the different nature of the different middleware components, the interfaces presented to the end users are highly heterogeneous, with APIs for different components belonging to different layers of abstraction. In LCG-2 [1], for example, there is no API to the information system apart from the standard LDAP C API, which is totally different from the R-GMA API [2]: this prevents applications from having a unified view of a mixed information system. This case will be thoroughly discussed in another contribution [3].

To bridge the gap between the existing middleware and the application level, a high-level technology independent interface must be defined. Such interface should be placed on top of the existing components in order to shield the end user from the underlying technology and offer a well defined common interface for the applications. In addition, the interface should be Grid-aware in the sense that it should be able to dynamically adapt to the particular runtime conditions in a heterogeneous environment like the Grid.

A good example are the security services, for which a standard, high-level interface to the various security mechanisms is also missing; we will focus our discussion on this case, describing a prototype for a generic handshaking and authentication interface, after having examined some existing ones.

EXISTING SECURITY INTERFACES

Several solutions have been adopted to implement security in distributed computing environments: the Grid Security Infrastructure (GSI) from Globus [4], which is based on public key encryption and X.509 certificates and adopts the standard GSS-API [5]; the EDG Authorization Framework [6], which allows to secure Web Services with GSI-like certificates; in more specific contexts, like XRootd [7], Kerberos, host-based and user/password authentication have been used; AFS security [8] itself may be used in a Grid. In the following, we will examine the commonalities of these mechanisms and see how they can be exploited to define an abstract, object-based interface to allow developers to interact transparently with many mechanisms.

The GSI model supports mutual authentication, using the SSL protocol. Each peer owns a certificate and a private key, issued by a certification authority (CA) trusted by the other peer. The initiator sends its certificate to the acceptor, which verifies the CA signature, and generates a "challenge" to ensure that the initiator owns its private key; the same is done in reverse to prove the acceptor's identity to the initiator.

This mechanism fits easily to the GSS-API concept,

where both peers, on the basis of their credentials, undergo an exchange of data which requires a certain number of iterations (irrelevant to the user) and whose outcome, if successful, is the establishment of a secure *context*. Once a secure context exists, the parties may use it to encrypt the exchanged data, or to guarantee its integrity. In fact, the GSS-API paradigm seems to be generic enough to cover all the functionalities commonly required in terms of security in a Grid.

We also examined the XRootd security interface [7], which uses Kerberos as mechanism and hides effectively the complexity of the Kerberos API. At the basis of the interface, there is the *XrdSecprotocol* class, which is instantiated by a client to contain the information about the secure context going to be created with a server. The method *getCredentials()* generates a ticket for a given server; the method *Authenticate()* is used by the server to decode the client credentials, and if it succeeds, it informs the client. Although this is done in a single pass, it is easy to adapt the API to mechanisms that require many exchanges of data, like in GSI, by calling repeatedly *getCredentials()* and *Authenticate()*.

The XRootd security API is much simpler than the GSS-API, and can be used with any authentication protocol at the application level. Nevertheless, it leaves to the user almost no way to configure the properties of the authentication procedure, or to inquire the properties of the peers' credentials or of the secure context. This is not a problem if the Xrootd API is used in a particular application, where every parameter can be defined statically in the code, but it prevents from using the same implementation with another application, which may require different choices of the parameters.

SECURITY INTERFACE PROTOTYPE

In this section, we describe an attempt to define a generic, object-oriented interface to a security service, with references to possible implementations, and a typical use case.

The basic idea is to allow an entity (referred to as the *client*), given its credentials, to try to establish a communication with a peer (referred to as the *server*), giving a list of security mechanisms in order of preference; the server chooses one and both peers load the corresponding library, where all the libraries share the same interface (see Fig. 1).

Authentication Interface

The class diagrams for the proposed interface are shown in Fig. 2. The *Principal* class is intended to represent a principal name, that is, a string which should identify an entity, like *username* or *service@hostname*; this is internally translated to something understood by the underlying security mechanism. A *type* can be given to specify the kind of principal (a user, a host-based service, an anonymous entity, etc.). Examples of principal names are */C=CH/O=CERN/OU=GRID/CN=John Doe* or *host@lxb0706.cern.ch.*

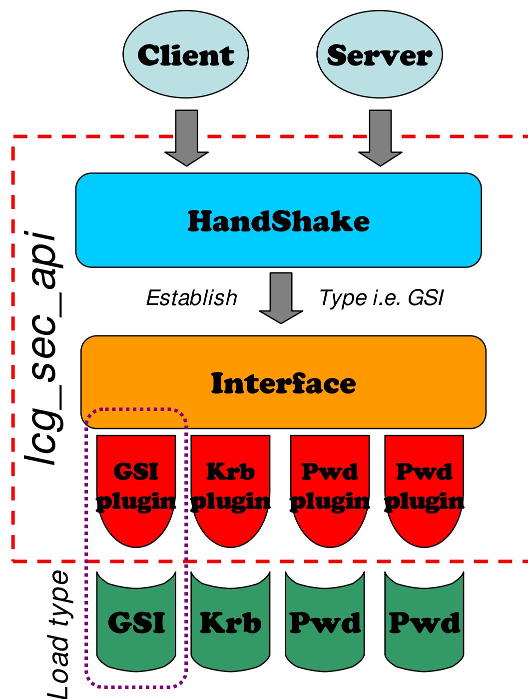

Figure 1: The relationships between client, server and the proposed interface.

The *Credential* class is meant to contain all the needed information about the entity's credentials (a proxy certificate, a Kerberos key, a password, etc.), the main data member being a pointer to the memory location containing the credential data. It is expected that the mechanism defines a default location for credentials and, if none exist when the constructor is invoked, they are created whenever possible (e.g. prompting for a password). In other words, there must be the concept of *default credentials*, which are created, or used if already existing, when a constructor is called without specifying a principal. Appropriate methods are provided to extract the principal from the credentials, the remaining lifetime of the credentials and if they are appropriate to initiate or to accept an authorization request.

The *Context* class is the core of the interface: it controls the authentication process and contains the all information about the security context, once this is established (for Kerberos, this would include the session key; for GSI, the certificate of the other party, etc.).

To perform the authentication, both peers instantiate a Context object using some credentials (maybe the default ones), and the principal of the other peer for the initiator of the authentication request. Then, the initiator will invoke the *sendRequest()* method to generate the data buffer to be sent to the other side, which will call the *recRequest()* method to interpret the received buffer and construct one to be sent to the client. The client will call again *sendRequest()* using as input the buffer received from

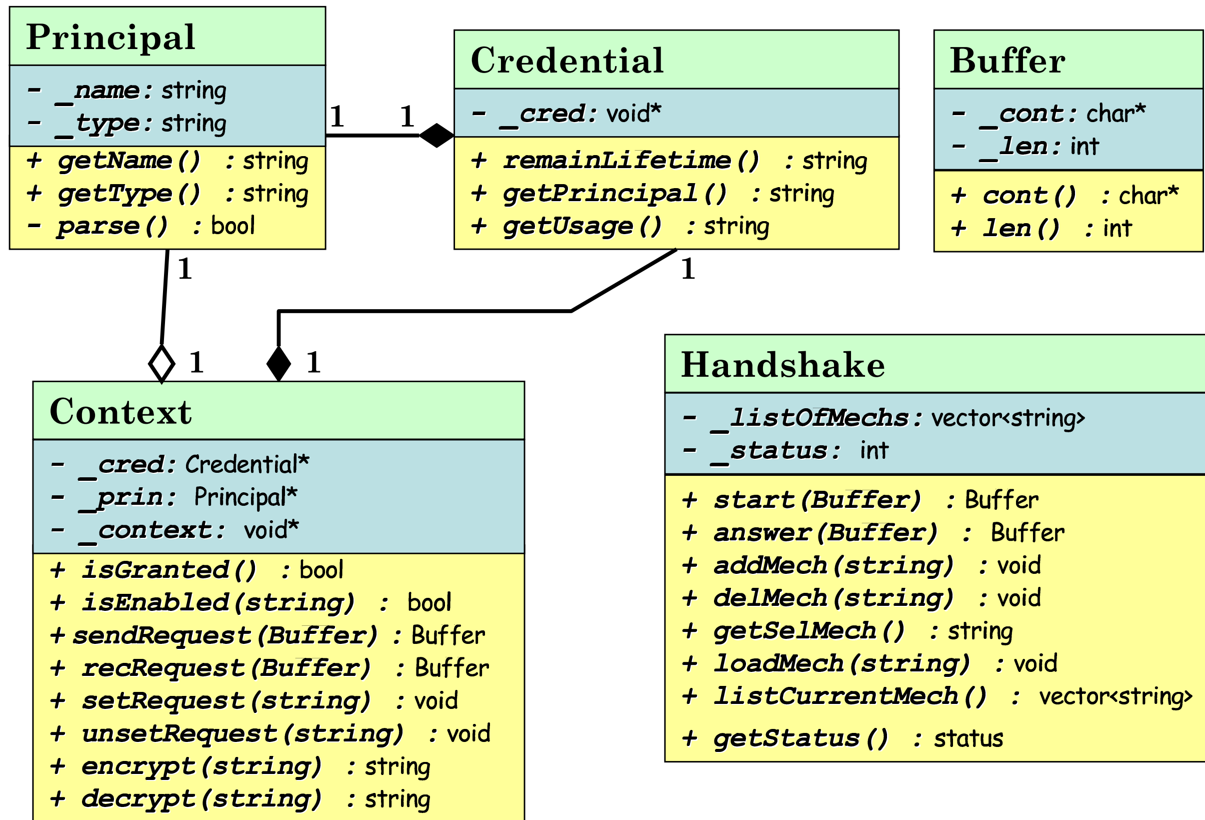

Figure 2: Class diagram.

the server and producing another one to be sent. The exchange of data ends when the mechanism-specific authentication algorithm ends; at any moment, the boolean method *isGranted()* tells if the authentication succeeded or not.

Before initiating the authentication request, the features of the context required by the client can be set with the *setRequest(feature)* method; such features may include the ability to encrypt the data, mutual authentication, delegation, etc. When the authentication succeeds, the method *isEnabled(feature)* tells if a particular feature is actually available. Finally, the methods *encrypt()* and *decrypt()* allow to use the context information (like a session key) to encrypt and decrypt a buffer.

Handshaking Interface

Before the authentication can start, the two entities must agree on a particular security mechanism; if they do so, the appropriate plug-in is loaded by both sides and the authentication begins. In this section, we propose an interface to this purpose (Fig. 2).

A class named *Handshake* is used to manage the process of choosing a security mechanism. The initiator must call a constructor with a list of preferred mechanisms, and similarly does the acceptor, with a list of supported mechanisms. The methods *addMech()* and *delMech()* can be used to add and remove mechanisms from the list. The initiator calls *start()* to prepare a buffer with the proposed mechanisms to be sent over to the acceptor, which will call *answer()* to interpret the buffer and create a new one with the selected mechanism (or a subset of the initial mechanisms). The method *getStatus()* returns an enum type whose value is *ONGOING*, *SUCCEEDED* or *FAILED*: when it returns *SUCCEEDED*, the method *getSelMech()* shall give the agreed mechanism. Finally, the method *loadMech()* loads the dynamic library specific to the given mechanism.

We wish to stress the fact that this interface does not impose any restrictions on the number of iterations needed for the handshaking. Concerning the handshaking algorithm itself, at each iteration it needs only to remove at least one mechanism from the list of mechanisms received from the other peer in order to ensure convergence.

An Example Use Case

The most obvious use case foresees that a client wants to establish a communication with a server knowing nothing of the kind of authentication required (if any). It will open a socket to a given port on the server host and start a handshaking with the security mechanisms it is capable or willing to use. If eventually the client and the server agree on a mechanism, they load the corresponding library and start the authentication process as already explained, provided of course that they hold valid credentials. An interaction diagram for this use case is shown in Fig. 3.

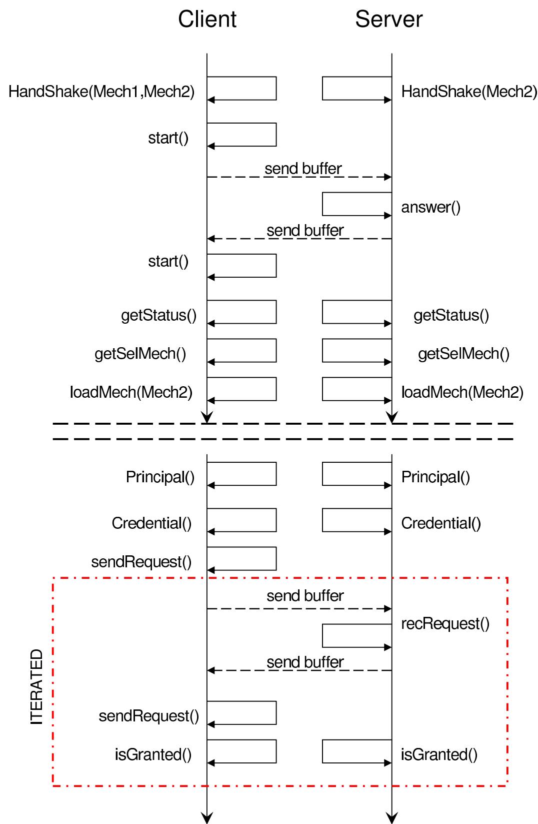

Figure 3: Interaction diagram for a typical use case.

Real Use Cases

Apart from the simple example of the previous section, the security interface that we are proposing could be adopted by current or future componentes of the LCG middleware.

The LCG data management tools could use data transfer protocols (other than GridFTP which requires by definition GSI) with whatever authentication mechanism (e.g. RFIO with Kerberos or GSI).

In addition to that, the new LCG prototype for software installation across sites [9] will benefit from the functionalities offered by the security high level interface. In fact, in such prototype, a client needs to interact with a server both in insecure and secure way, depending on the action it needs to perform; using the LCG security API, the client could negotiate with the server at run time the type of security context it requires, rather than having to run two separates instances of the service, one in secure (GSI) and the other in insecure mode.

CONCLUSIONS AND FUTURE WORK

We have presented a generic programming interface to an authentication service for network connections in a Grid environment, with the purpose to hide from the user the specific security mechanism used. This high level API in fact can accommodate quite different authentication methods (like password-based, Kerberos, GSI ...) and provide a unique interface for the user. In addition, the interface is grid-aware, in the sense that it able to dynamically load the different components, depending on the runtime requirements. The final product should provide all the functionalities of the existing interfaces described at the beginning of this document (XRootd, GSS-API, etc ...) and should be complete enough to satisfy the use cases mentioned above. Our plans foresee to write a number of implementations, starting with a password based and a GSI-like mechanism, and investigate how it could be extended to other domains, like authorization.

ACKNOWLEDGMENTS

This work has been founded by the Istituto nazionale di Fisica Nucleare, Roma, Italy and the Ministerio de Educación y Ciencia, Madrid, Spain.

REFERENCES

[1] https://edms.cern.ch/file/454439//LCG-2-UserGuide.html/.

[2] http://www.r-gma.org/.

[3] A. Delgado Peris *et al.*, “Experience integrating a General Information System API in LCG Job Management and Monitoring Services”, these CHEP proceedings.

[4] http://www.globus.org/.

[5] J. Linn, “RFC 2743 - Generic Security Service Application Program Interface Version 2, Update 1”.

[6] http://edg-wp2.web.cern.ch/edg-wp2/security/edg-java-security.html.

[7] A. Hanushevsky, H. Stockinger, “A Proxy Service for the xrootd Data Server”, to be published on the Proceedings of the First International Workshop on Scientific Applications on Grid Computing (SAG’04), Beijing, China, 20-24 September 2004.

[8] http://www.openafs.org/.

[9] R. Santinelli *et al.*, “Experiment Software Installation experience in LCG-2”, these CHEP proceedings.